JH_task3

November 30, 2022

1 Assignment 3 - Sparse matrices

This assignment makes up 30% of the overall marks for the course. The deadline for submitting this assignment is **5pm on Thursday 1 December 2022**.

Coursework is to be submitted using the link on Moodle. You should submit a single pdf file containing your code, the output when you run your code, and your answers to any text questions included in the assessment. The easiest ways to create this file are:

- Write your code and answers in a Jupyter notebook, then select File -> Download as -> PDF via LaTeX (.pdf).
- Write your code and answers on Google Colab, then select File -> Print, and print it as a pdf.

Tasks you are required to carry out and questions you are required to answer are shown in bold below.

1.1 Part 1: Implementing a CSR matrix

Scipy allows you to define your own objects that can be used with their sparse solvers. You can do this by creating a subclass of `scipy.sparse.LinearOperator`. In the first part of this assignment, you are going to implement your own CSR matrix format.

[4]:
```python
from scipy.sparse.linalg import LinearOperator
from scipy.sparse import coo_matrix
from scipy.sparse import csr_matrix
import numba
import numpy as np
%matplotlib inline
from matplotlib import pyplot as plt
from matplotlib import cm
import seaborn as sns
plt.style.use('seaborn-notebook')
```

1.1.1 a)

Make a copy of this code snippet and **implement the methods `__init__`, `__add__` and `matvec`.** The method `__init__` takes a COO matrix as input and will initialise the CSR matrix: it currently includes one line that will store the shape of the input matrix. You should add code here that converts from COO to CSR format and stores the appropriate data for the CSR matrix. The method `__add__` will overload `+` and so allow you to add two of your CSR matrices together. The method `matvec` will define a matrix-vector product: Scipy will use this when you tell it to use a sparse solver on your operator.

```
[9]: class CSRMatrix(LinearOperator):

         def __init__(self, coo_matrix):

             self.shape = coo_matrix.shape
             self.dtype = coo_matrix.dtype
             self.data = np.zeros(len(coo_matrix.data))
             self.indices = np.zeros(len(coo_matrix.data), dtype=int)
             self.indptr = np.zeros(coo_matrix.shape[0]+1, dtype=int)

             for i in range(coo_matrix.shape[0]):

                 #Find the number of entries the coo matrix has stored for the given␣
      ↪row
                 #Save the pointers
                 self.indptr[i+1] = np.sum(coo_matrix.row == i) + self.indptr[i]

                 #Save the indices and data values
                 self.data[self.indptr[i]:self.indptr[i+1]] = coo_matrix.
      ↪data[coo_matrix.row == i]
                 self.indices[self.indptr[i]:self.indptr[i+1]] = coo_matrix.
      ↪col[coo_matrix.row == i]


         def __add__(self, other):

             m, n = self.shape
             output = np.zeros((m, n), dtype=np.float64)
             for row_index in range(m):

                 #Find where the row data for the first matrix is stored
                 col_start_self = self.indptr[row_index]
                 col_end_self = self.indptr[row_index+1]

                 #Add the data values to the row at the corresponding columns
                 for col_index in range(col_start_self, col_end_self):
                     output[row_index, self.indices[col_index]] += self.
      ↪data[col_index]

                 #Find where the row data for the second matrix is stored
                 col_start_other = other.indptr[row_index]
                 col_end_other = other.indptr[row_index+1]
```

```
            #Add the data values to the row at the corresponding columns
            for col_index in range(col_start_other, col_end_other):
                output[row_index, other.indices[col_index]] += other.
↪data[col_index]

        #I could now convert the output into a COO matrix and pass it into␣
↪CSRMatrix() - you mentioned
        #in one of the problem classes that it was okay to output a dense␣
↪matrix as long as I didn't convert
        #the CSR matrices to dense while doing the addition
        return output

    def _matvec(self, x):

        y = np.zeros(self.shape[0], dtype=np.float64)
        for row_index in range(self.shape[0]):

            #Find the start and the end of the row's data
            col_start = self.indptr[row_index]
            col_end = self.indptr[row_index + 1]

            #Multiply the non zero columns with the corresponding values in the␣
↪vector then sum
            for col_index in range(col_start, col_end):
                y[row_index] += self.data[col_index] * x[self.
↪indices[col_index]]

        return y
```

1.1.2 b)

Write tests to check that the `__add__` and `matvec` methods that you have written are correct. These test should use appropriate `assert` statements.

Testing __add

[10]:
```
non_zero = 6
shape = (4,4)

for i in range(20):

    row1  = np.random.randint(0,shape[0],non_zero)
    col1  = np.random.randint(0,shape[1],non_zero)
    data1 = np.random.randn(non_zero)

    row2  = np.random.randint(0,shape[0],non_zero)
    col2  = np.random.randint(0,shape[1],non_zero)
    data2 = np.random.uniform(0,10,non_zero)

    A_coo = coo_matrix((data1, (row1, col1)), shape=shape)
    B_coo = coo_matrix((data2, (row2, col2)), shape=shape)

    my_csr_A = CSRMatrix(A_coo)
    my_csr_B = CSRMatrix(B_coo)

    assert np.allclose(my_csr_A + my_csr_B, A_coo.toarray() + B_coo.toarray())
print("__add__ matched")

print(f" My addition: \n {my_csr_A + my_csr_B}")
print(f" Coo addition: \n {A_coo.toarray() + B_coo.toarray()}")
print(f"Difference:  {np.sum((my_csr_A + my_csr_B) - (A_coo.toarray() + B_coo.
 ↪toarray()))}")
```

```
__add__ matched
 My addition:
 [[ 0.          -0.84822817  0.           2.53618116]
 [ 0.3947539   2.92181824  6.42297355  0.        ]
 [10.52300802  0.          0.          0.        ]
 [ 0.2820623   0.          8.76758847  2.24155102]]
 Coo addition:
 [[ 0.          -0.84822817  0.           2.53618116]
 [ 0.3947539   2.92181824  6.42297355  0.        ]
 [10.52300802  0.          0.          0.        ]
 [ 0.2820623   0.          8.76758847  2.24155102]]
Difference:  0.0
```

Testing __matvec

[11]:
```
non_zeros = 6
shape = (4,4)
```

[12]:
```
for i in range(20):

    #Get random
    row  = np.random.randint(0,shape[0],non_zero)
    col  = np.random.randint(0,shape[1],non_zero)
    data = np.random.randn(non_zero)

    A_coo = coo_matrix((data, (row, col)), shape=shape)

    my_csr = CSRMatrix(A_coo)
    vector = np.random.randn(shape[0])

    assert np.allclose(my_csr*vector, A_coo@vector)
```

```
print("_matvect matched")

print(f" My multiplication: \n {my_csr*vector}")
print(f" Coo multiplication: \n {A_coo@vector}")
print(f"Difference:  {np.sum((my_csr*vector) - (A_coo@vector))}")
```

```
_matvect matched
 My multiplication:
 [-0.66208343 -0.05245534  0.41661614  0.13463015]
 Coo multiplication:
 [-0.66208343 -0.05245534  0.41661614  0.13463015]
Difference:  0.0
```

1.1.3 c)

For a collection of sparse matrices of your choice and a random vector, **measure the time taken to perform a `matvec` product**. Convert the same matrices to dense matrices and **measure the time taken to compute a dense matrix-vector product using Numpy**. **Create a plot showing the times of `matvec` and Numpy for a range of matrix sizes** and **briefly (1-2 sentence) comment on what your plot shows**.

[13]:
```
from time import perf_counter
```

[14]:
```
density = 0.1

dense_times = []
sparse_times = []
for i in range(10,500):

    shape = (i,i)

    #Increase the number of non zero entries as the size of the matrix grows
    non_zero = int(i*i*density)

    row  = np.random.randint(0,shape[0],non_zero)
    col  = np.random.randint(0,shape[1],non_zero)
    data = np.random.randn(non_zero)

    A_coo = coo_matrix((data, (row, col)), shape=shape)
    A = A_coo.toarray()

    my_csr = CSRMatrix(A_coo)
    vector = np.random.randn(shape[0])

    start = perf_counter()
    dense_sol = A@vector
    dense_times.append(perf_counter()-start)
    start = perf_counter()
    sparse_sol = my_csr*vector
    sparse_times.append(perf_counter()-start)

    assert np.allclose(dense_sol, sparse_sol)
```

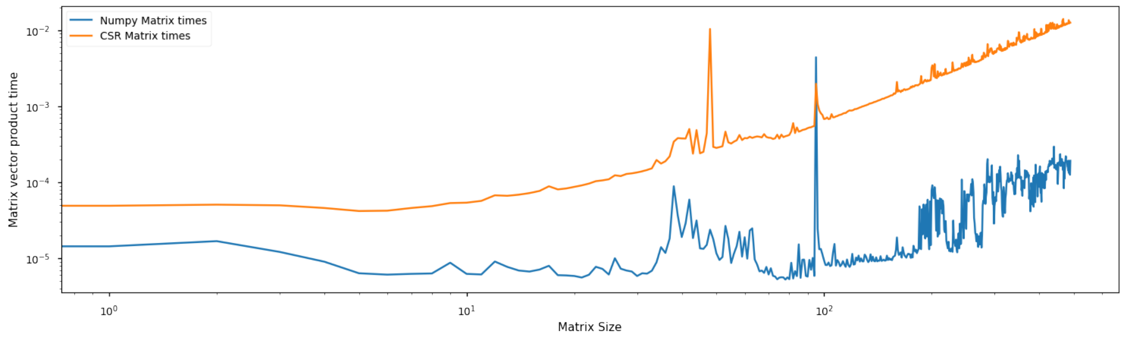
[15]:
```
fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)

ax.plot(dense_times, label = 'Numpy Matrix times')
ax.plot(sparse_times, label = 'CSR Matrix times')

ax.set_xlabel("Matrix Size")
ax.set_ylabel("Matrix vector product time")

ax.set_xscale("log")
ax.set_yscale("log")

plt.legend();
```



Answer The dense matrix vector product appears to be faster than the sparse matrix vector product. This was expected because the numpy function is highly optimized. However, I would expect that as the size of the matrices grow, my sparse matrix vector product would become faster than the numpy function because it is having to do fewer operations.

As shown below, if the density of the matrices is reduced, the sparse matrix vector product overtakes the dense vector product at smaller matrix sizes.

[16]:
```
density = 0.001

dense_times = []
sparse_times = []
for i in range(10,1000):

    shape = (i,i)
    #Increase the number of non zero entries as the size of the matrix grows
    non_zero = int(i*i*density)

    row  = np.random.randint(0,shape[0],non_zero)
    col  = np.random.randint(0,shape[1],non_zero)
    data = np.random.randn(non_zero)

    A_coo = coo_matrix((data, (row, col)), shape=shape)
    A = A_coo.toarray()

    my_csr = CSRMatrix(A_coo)
    vector = np.random.randn(shape[0])

    start = perf_counter()
    dense_sol = A@vector
    dense_times.append(perf_counter()-start)

    start = perf_counter()
    sparse_sol = my_csr*vector
    sparse_times.append(perf_counter()-start)


    assert np.allclose(dense_sol, sparse_sol)
```

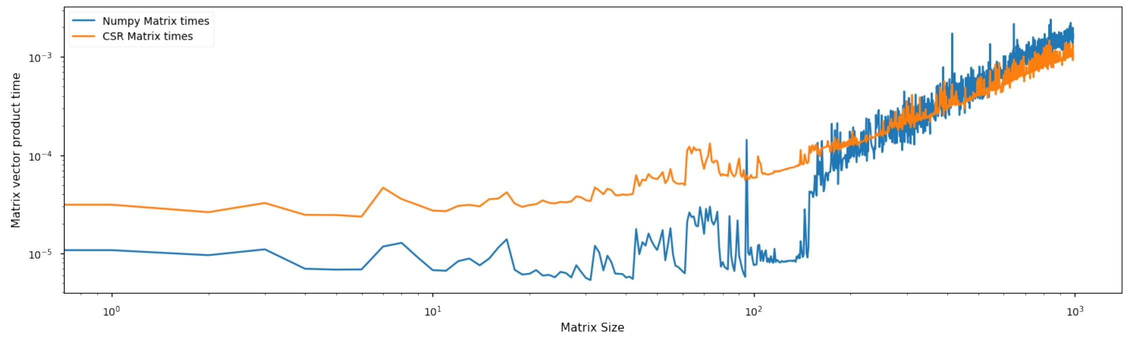
[17]:
```
fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)

ax.plot(dense_times, label = 'Numpy Matrix times')
ax.plot(sparse_times, label = 'CSR Matrix times')

ax.set_xlabel("Matrix Size")
ax.set_ylabel("Matrix vector product time")

ax.set_xscale("log")
ax.set_yscale("log")

plt.legend();
```



1.1.4 d)

For a matrix of your choice and a random vector, **use Scipy's gmres and cg sparse solvers to solve a matrix problem using your CSR matrix**. Check if the two solutions obtained are the same. **Briefly comment (1-2 sentences) on why the solutions are or are not the same.**

[18]:
```python
from scipy.sparse.linalg import gmres
from scipy.sparse.linalg import cg
```

[19]:
```python
non_zeros = 20
shape = (4,4)
row  = np.random.randint(0,shape[0],non_zero)
col  = np.random.randint(0,shape[1],non_zero)
data = np.random.uniform(0,10,non_zero)

A_coo = coo_matrix((data, (row, col)), shape=shape)

my_csr = CSRMatrix(A_coo)
vector = np.random.randint(0,10,shape[0])
```

[20]:
```python
gmres_sol =  gmres(my_csr, vector)
print(f"gmres solution: {gmres_sol}")
```

```
gmres solution: (array([ 0.04735473, -0.04992373, -0.02156769,  0.03343227]), 0)
```

[21]:
```python
cg_sol =  cg(my_csr, vector)
print(f"cg solution: {cg_sol}")
```

```
cg solution: (array([ 0.17090844,  0.10408788, -0.48252081,  0.27029173]), 40)
```

[22]:
```python
print(f"Solution Difference: {np.linalg.norm(gmres_sol[0]-cg_sol[0])} ")
```

```
Solution Difference: 0.5545856811268067
```

Answer In general the answers do not agree. gmres appears to solve the problem whereas cg ends up with a large residual. This is because cg requires the matrix to be positive definite - we haven't set this to be the case so cg is just producing random noise and not solving the system.

If we now try and use a positive definite matrix:

[23]:
```python
from numpy.random import RandomState
```

[24]:
```python
n = 5

rand = RandomState(0)
```

```
Q, _ = np.linalg.qr(rand.randn(n, n))
D = np.diag(rand.rand(n))
A = Q.T @ D @ Q
```

[25]:
```
my_csr = CSRMatrix(coo_matrix(A))
vector = np.random.randint(0,10,n)
```

[26]:
```
gmres_sol =  gmres(my_csr, vector)
print(f"gmres solution: {gmres_sol}")
```

```
gmres solution: (array([ 93.38451432,  43.04920454, 104.96731727, -70.48068141,
         7.30532889]), 0)
```

[27]:
```
cg_sol =  cg(my_csr, vector)
print(f"cg solution: {cg_sol}")
```

```
cg solution: (array([ 93.38451432,  43.04920454, 104.96731727, -70.48068141,
         7.30532889]), 0)
```

[28]:
```
print(f"Solution Difference: {np.linalg.norm(gmres_sol[0]-cg_sol[0])} ")
```

```
Solution Difference: 8.515889703807831e-13
```

Both methods now obtain a residual of zero and have solutions which are very close.

1.2 Part 2: Implementing a custom matrix

Let A by a $2n$ by $2n$ matrix with the following structure:

- The top left n by n block of A is a diagonal matrix
- The top right n by n block of A is zero
- The bottom left n by n block of A is zero
- The bottom right n by n block of A is dense (but has a special structure defined below)

In other words, A looks like this, where $*$ represents a non-zero value

$$A = \begin{pmatrix} * & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & * & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & * & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & * & 0 & 0 & \cdots & 0 \\ \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \end{pmatrix}$$

Let $\tilde{A}$ be the bottom right n by n block of A. For some matrices $\tilde{A}$ can be written as

$$\tilde{A} = TW,$$

where T is a n by 2 matrix (a tall matrix); and where W is a 2 by n matrix (a wide matrix).

1.2.1 a)

Implement a Scipy `LinearOperator` for matrices of this form. Your implementation must include a matrix-vector product (`matvec`) and the shape of the matrix (`self.shape`), but does not need to include and `__add__` function. In your implementation of `matvec`, you should be careful to ensure that the product does not have more computational complexity then necessary.

[29]:
```
class CustomMatrix(LinearOperator):

    def __init__(self, diagonal, T, W):

        self.shape = (2*len(diagonal), 2*len(diagonal))
        self.dtype = 'float64'
        self.diagonal = diagonal
        self.submatrix = T@W

    def _matvec(self, x):

        y = np.zeros(self.shape[0])

        #Multiply the first half of the vector by the diagonal elements
        y[:int(self.shape[0]/2)] = self.diagonal*x[:int(self.shape[0]/2)]

        #Multiply the second half of the vector by the sub matrix
        y[int(self.shape[0]/2):] = self.submatrix@x[int(self.shape[0]/2):]

        return y

    def toarray(self):

        #Output the custom matrix in dense form
        return np.block([[np.diag(self.diagonal),np.zeros((len(self.
↪diagonal),len(self.diagonal))) ],
                          [np.zeros((len(self.diagonal),len(self.diagonal))) ,␣
↪self.submatrix       ]])
```

1.2.2 b)

For a range of values of n, **create matrices where the entries on the diagonal of the top-left block and in the matrices** T **and** W **are random numbers.** For each of these matrices, **compute matrix-vector products using your implementation and measure the time taken to compute these.** Create dense versions of the matrices, and **measure the time taken by Numpy to compute matrix-vector products. Make a plot showing time taken against** n**. Comment (2-4 sentences) on what your plot shows, and why you think one**

of these methods is faster than the other (or why they take the same amount of time if this is the case).

Below is a small test to make sure my class is working correctly:

[30]:
```
n = 3

diag = np.random.rand(n)
T = np.random.rand(n,2)
W = np.random.rand(2,n)

custom_matrix = CustomMatrix(diag, T, W)

dense_matrix = custom_matrix.toarray()
```

[31]:
```
print("Dense Matrix:")
print(dense_matrix)
print()
print("Custom matrix diagonal:")
print(custom_matrix.diagonal)
print()
print("Custom matrix TW:")
print(custom_matrix.submatrix)
```

```
Dense Matrix:
[[0.88340187 0.         0.         0.         0.         0.        ]
 [0.         0.39869217 0.         0.         0.         0.        ]
 [0.         0.         0.75225417 0.         0.         0.        ]
 [0.         0.         0.         0.17941229 0.1702187  0.16804908]
 [0.         0.         0.         0.92571224 0.85736977 0.84439913]
 [0.         0.         0.         0.68218888 0.74592084 0.74605523]]

Custom matrix diagonal:
[0.88340187 0.39869217 0.75225417]

Custom matrix TW:
[[0.17941229 0.1702187  0.16804908]
 [0.92571224 0.85736977 0.84439913]
 [0.68218888 0.74592084 0.74605523]]
```

[32]:
```
dense_times = []
sparse_times = []
ns = np.linspace(10,2000,100).astype(int)

for n in ns:

    diag = np.random.rand(n)
    T = np.random.rand(n,2)
    W = np.random.rand(2,n)

    custom_matrix = CustomMatrix(diag, T, W)

    dense_matrix = custom_matrix.toarray()

    vector = np.random.rand(2*n)


    start = perf_counter()
    dense_sol = dense_matrix@vector
    dense_times.append(perf_counter()-start)

    start = perf_counter()
    sparse_sol = custom_matrix*vector
    sparse_times.append(perf_counter()-start)


    assert np.allclose(dense_sol, sparse_sol)
```

[33]:
```
print(f" Sparse multiplication: \n {custom_matrix*vector}")
print(f" Dense multiplication: \n {dense_matrix@vector}")
print(f"Difference:  {np.sum((custom_matrix*vector) - (dense_matrix@vector))}")
```

```
 Sparse multiplication:
 [3.19388208e-01 1.74321303e-01 2.51475435e-02 … 6.73213888e+02
 6.50460013e+02 1.96775336e+02]
 Dense multiplication:
 [3.19388208e-01 1.74321303e-01 2.51475435e-02 … 6.73213888e+02
 6.50460013e+02 1.96775336e+02]
Difference:  0.0
```

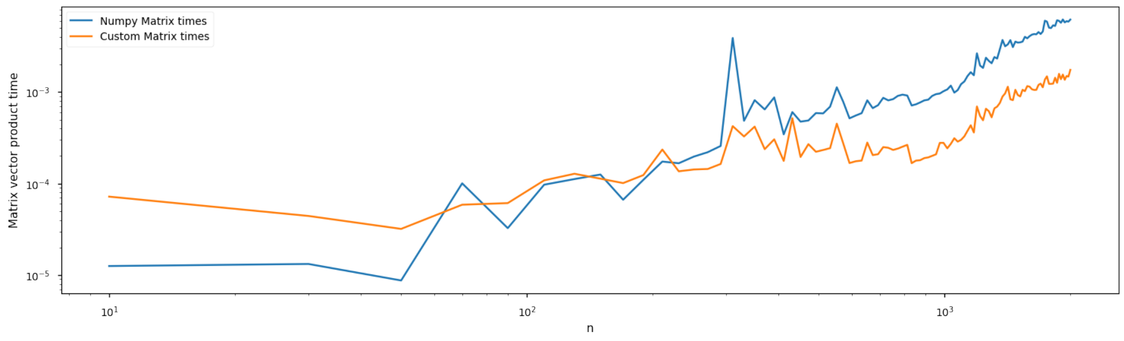
[34]:
```
fig, ax = plt.subplots(1,figsize=(18, 5),dpi=100)

ax.plot(ns, dense_times, label = 'Numpy Matrix times')
ax.plot(ns, sparse_times, label = 'Custom Matrix times')

ax.set_xlabel("n")
ax.set_ylabel("Matrix vector product time")

ax.set_xscale("log")
ax.set_yscale("log")

plt.legend();
```



Answer Similar to the previous section, the dense multiplication starts out faster than the sparse multiplication as numpy is highly optimized. However, the sparse multiplication quickly overtakes the dense multiplication as the dimensions of the problem grow larger. This is because for a given value of n the dense multiplication we do $(2n)^2$ multiplications, for the custom format we are doing n multiplications for the diagonal times the top half of the vector and n^2 operations for TW times the bottom half of the vector so $n + n^2$ multiplications. Therefore the number of operations the custom method is performing grows at a slower rate when compared to the dense method.

– $n + n^2 \leq (2n)^2$ for $n \geq \frac{1}{3}$